CHAPTER 10

# Sparse Matrices and Graphs

We have gone through numerous examples of arrays and matrices that are essential in many aspects of numerical computing. And, we have represented arrays with the NumPy `ndarray` data structure, a heterogeneous representation that stores all the array elements it represents. This is often the most efficient way to represent an object, such as a vector, matrix, or higher-dimensional array. However, notable exceptions are matrices where most of the elements are zeros. Such matrices are known as *sparse matrices*, and they occur in many applications, for example, in connection networks (such as circuits) and large algebraic equation systems that arise, for example, when solving partial differential equations (see Chapter 11 for examples).

For matrices dominated by zero elements, storing all the zeros in the computer's memory is inefficient, and it is more suitable to store only the nonzero values with additional information about their locations. For non-sparse matrices, known as *dense matrices*, such a representation is less efficient than storing all values consecutively in the memory, but for large sparse matrices, it can be vastly superior.

There are several options for working with sparse matrices in Python. This chapter focuses on the sparse matrix module in SciPy, `scipy.sparse`, which provides a feature-rich and easy-to-use interface for representing sparse matrices and carrying out linear algebra operations on such objects. Another option is PySparse,[1] which offers similar functionality. For very large-scale problems, the PyTrilinos[2] and PETSc[3] packages have powerful parallel implementations of many sparse matrix operations. However, using these packages requires more programming, and they have a steeper learning curve and are more difficult to install and set up. For most basic use cases, SciPy's `sparse` module is the most suitable option or at least an appropriate starting point.

The end of the chapter briefly explores representing and processing graphs using the SciPy `sparse.csgraph` module and the NetworkX library. Graphs can be represented as adjacency matrices, which in many applications are very sparse. Graphs and sparse matrices are, therefore, closely connected topics.

## Importing Modules

The main module we work with in this chapter is the `sparse` module in the SciPy library. Let's assume this module is included under the name `sp`, and we also need to explicitly import its `linalg` submodule to make it accessible through `sp.linalg`.

```
In [1]: import scipy.sparse as sp
In [2]: import scipy.sparse.linalg
```

[1] http://pysparse.sourceforge.net
[2] http://trilinos.org/packages/pytrilinos
[3] See http://www.mcs.anl.gov/petsc and https://bitbucket.org/petsc/petsc4py for its Python bindings.


R. Johansson, *Numerical Python*, https://doi.org/10.1007/979-8-8688-0413-7_10

We also need the NumPy library, which we, as usual, import under the name `np`, and the Matplotlib library for plotting:

```
In [3]: import numpy as np
In [4]: import matplotlib.pyplot as plt
```

The last part of this chapter uses the `networkx` module, which we import under the name `nx`.

```
In [5]: import networkx as nx
```

# Sparse Matrices in SciPy

The basic idea of sparse matrix representation is to avoid storing the excessive zeros in a sparse matrix. In dense matrix representation, where all elements of an array are stored consecutively, it is sufficient to store the values since each element's row and column indices are implicitly known from the position in the array. However, if we store only the nonzero elements, we must also store each element's row and column indices. There are numerous approaches to organizing the storage of the nonzero elements and their corresponding row and column indices. These approaches have different advantages and disadvantages, for example, how easy it is to create the matrices and, perhaps more importantly, how efficiently they can be used to implement mathematical operations on the sparse matrices. A summary and comparison of sparse matrix formats available in the SciPy `sparse` module is given in Table 10-1.

***Table 10-1.*** *Summary and Comparison of Methods to Represent Sparse Matrices*

| Type | Description | Pros | Cons |
|---|---|---|---|
| Coordinate list (COO, `sp.coo_matrix`) | Nonzero values are stored in a list together with their row and column. | Simple to construct and efficient to add new elements. | Inefficient element access. Not suitable for mathematical operations, such as matrix multiplication. |
| List of lists (LIL, `sp.lil_matrix`) | Stores a list of column indices for nonzero elements for each row and a list of the corresponding values. | Supports slicing operations. | Not ideal for mathematical operations. |
| Dictionary of keys (DOK, `sp.dok_matrix`) | Nonzero values are stored in a dictionary with a tuple of (row, column) as key. | Simple to construct and fast to add, remove and access elements. | Not ideal for mathematical operations. |
| Diagonal matrix (DIA, `sp.dia_matrix`) | Stores lists of diagonals of the matrix. | Efficient for diagonal matrices. | Not suitable for nondiagonal matrices. |
| Compressed sparse column (CSC, `sp.csc_matrix`) and compressed sparse row (CSR, `sp.csr_matrix`) | Stores the values together with arrays with column or row indices. | Relatively complicated to construct. | Efficient matrix-vector multiplication. |
| Block-sparse matrix (BSR, `cp.bsr_matrix`) | Similar to CSR, but for sparse matrices with dense submatrices. | Efficient for their specific intended purpose. | Not suitable for general-purpose use. |

A simple and intuitive approach for storing sparse matrices is storing lists with column indices and row indices together with the list of nonzero values. This format is called *coordinate list format*, abbreviated as COO in SciPy. The `sp.coo_matrix` class represents sparse matrices in this format. This format is straightforward to initialize. For instance, with the matrix

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 \\ 4 & 0 & 0 & 0 \end{bmatrix},$$

we can easily identify the nonzero values $[A_{01} = 1, A_{13} = 2, A_{22} = 3, A_{30} = 4]$ and their corresponding rows [0, 1, 2, 3] and columns [1, 3, 2, 0] (note that here we have used Python's zero-based indexing). To create a `sp.coo_matrix` object, we can create lists (or arrays) for the values, row indices, and column indices and pass them to `sp.coo_matrix`. Optionally, we can also specify the shape of the array using the `shape` argument, which is useful when the nonzero elements do not span the entire matrix (for example, if there are columns or rows containing only zeros so that the shape cannot be correctly inferred from the row and column arrays).

```
In [6]: values = [1, 2, 3, 4]
In [7]: rows = [0, 1, 2, 3]
In [8]: cols = [1, 3, 2, 0]
In [9]: A = sp.coo_matrix((values, (rows, cols)), shape=[4, 4])
In [10]: A
Out[10]: <4x4 sparse matrix of type '<type 'numpy.int64'>'
              with 4 stored elements in Coordinate format>
```

The result is a data structure that represents the sparse matrix. All sparse matrix representations in SciPy's `sparse` module share several common attributes, many of which are derived from NumPy's `ndarray` object. Examples of such attributes are `size`, `shape`, `dtype`, and `ndim`, and common to all sparse matrix representations are the `nnz` (number of nonzero elements) and `data` (the nonzero values) attributes.

```
In [11]: A.shape, A.size, A.dtype, A.ndim
Out[11]: ((4, 4), 4, dtype('int64'), 2)
In [12]: A.nnz, A.data
Out[12]: (4, array([1, 2, 3, 4]))
```

In addition to the shared attributes, each type of sparse matrix representation also has attributes specific to storing the positions for each nonzero value. For `sp.coo_matrix` objects, there are `row` and `col` attributes for accessing the underlying row and column arrays.

```
In [13]: A.row
Out[13]: array([0, 1, 2, 3], dtype=int32)
In [14]: A.col
Out[14]: array([1, 3, 2, 0], dtype=int32)
```

Many methods are also available for operating on sparse matrix objects. Many of these methods are for applying mathematical functions on the matrix, for example, elementwise math methods like `sin`, `cos`, `arcsin`, and so on; aggregation methods like `min`, `max`, `sum`, and so on; mathematical array operators such as conjugate (`conj`) and transpose (`transpose`), and so on; and `dot` for computing the dot product between sparse matrices or a sparse matrix and a dense vector (the * operator also denotes matrix multiplication

for sparse matrices). For further details, see the docstring for the sparse matrix classes (summarized in Table 10-1). Another important family of methods is used to convert sparse matrices between different formats: for example, `tocoo`, `tocsr`, `tolil`, and so on. There are also methods for converting a sparse matrix to NumPy `ndarray` and NumPy `matrix` objects (i.e., dense matrix representations): `toarray` and `todense`.

For example, to convert the sparse matrix `A` from COO format to CSR format and a NumPy array, respectively, we can use the following.

```
In [15]: A.tocsr()
Out[15]: <4x4 sparse matrix of type '<type 'numpy.int64'>'
         with 4 stored elements in Compressed Sparse Row format>
In [16]: A.toarray()
Out[16]: array([[0, 1, 0, 0],
                [0, 0, 0, 2],
                [0, 0, 3, 0],
                [4, 0, 0, 0]])
```

The obvious way to access elements in a matrix, which has been used in numerous different contexts so far, is using the indexing syntax (e.g., `A[1,2]`), the slicing syntax (e.g., `A[1:3, 2]`), and so on. We can often use this syntax with sparse matrices, too, but not all representations support indexing and slicing, and if it is supported, it may not be an efficient operation. In particular, assigning values to zero-valued elements can be costly, as it may require rearranging the underlying data structures, depending on the format used. To incrementally add new elements to a sparse matrix, the LIL (`sp.lil_matrix`) format is a suitable choice, but this format is, on the other hand, not ideal for arithmetic operations.

When working with sparse matrices, it is common to face the situation that different tasks—such as construction, updating, and arithmetic operations—are most efficiently handled in other formats. Converting between different sparse formats is relatively efficient, so switching between different formats in different parts of an application is useful. Therefore, efficient use of sparse matrices requires understanding how different formats are implemented and what they are suitable for. Table 10-1 briefly summarizes the pros and cons of the sparse matrix formats available in SciPy's `sparse` module, and using the conversion methods, it is easy to switch between different formats. For a more in-depth discussion of the merits of the various formats, see the "Sparse Matrices"[4] section in the SciPy reference manual.

For computations, the most important sparse matrix representations in SciPy's `sparse` module are the CSR (Compressed Sparse Row) and CSC (Compressed Sparse Column) formats because they are well-suited for efficient matrix arithmetic and linear algebra applications. Other formats, like COO, LIL, and DOK, are mainly used for constructing and updating sparse matrices. Once a sparse matrix is ready for computations, it is best to convert it to CSR or CSC format, utilizing the `tocsr` or `tocsc` methods, respectively.

In the CSR format, the nonzero values (`data`) are stored along with an array that contains the column indices of each value (`indices`) and another array that stores the offsets of the column index array of each row (`indptr`). For instance, consider the following matrix:

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 5 & 6 \\ 7 & 0 & 8 & 9 \end{bmatrix}.$$

Here the nonzero values are [1, 2, 3, 4, 5, 6, 7, 8, 9] (`data`), and the *column indices* corresponding to the nonzero values in the first row are [0, 1], the second row [1, 2], the third row [2, 3], and the fourth row [0, 2, 3]. Concatenating these column index lists gives the `indices` array [0, 1, 1, 2, 2, 3, 0, 2, 3]. To keep track of which

[4] http://docs.scipy.org/doc/scipy/reference/sparse.html

row entries in this column index array correspond to, we can store the starting position for each row in a second array. The column indices of the first row are elements 0 to 1, the second-row elements 2 to 3, the third-row elements 4 to 5, and finally the fourth-row elements 6 to 9. Collecting the starting indices in an array gives [0, 2, 4, 6]. For convenience in the implementation, we also add the total number of nonzero elements at the end of this array, which results in the `indptr` array [0, 2, 4, 6, 9]. The following code creates a dense NumPy array corresponding to matrix $A$, converts it to a CSR matrix using `sp.csr_matrix`, and then displays the `data`, `indices`, and `indptr` attributes.

```
In [17]: A = np.array([[1, 2, 0, 0], [0, 3, 4, 0], [0, 0, 5, 6], [7, 0, 8, 9]])
    ...: A
Out[17]: array([[1, 2, 0, 0],
                [0, 3, 4, 0],
                [0, 0, 5, 6],
                [7, 0, 8, 9]])
In [18]: A = sp.csr_matrix(A)
In [19]: A.data
Out[19]: array([1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int64)
In [20]: A.indices
Out[20]: array([0, 1, 1, 2, 2, 3, 0, 2, 3], dtype=int32)
In [21]: A.indptr
Out[21]: array([0, 2, 4, 6, 9], dtype=int32)
```

With this storage scheme, the nonzero elements in the row with index `i` are stored in the `data` array between index `indptr[i]` and `indptr[i+1]-1`, and the column indices for these elements are stored at the same indices in the `indices` array. For example, the elements in the third row, with index `i=2`, start at `indptr[2]=4` and end at `indptr[3]-1=5`, which gives the element values, `data[4]=5` and `data[5]=6`, and the column indices, `indices[4]=2` and `indices[5]=3`. Thus, $A[2, 2] = 5$ and $A[2, 3] = 6$ (in zero indexbased notation).

```
In [22]: i = 2
In [23]: A.indptr[i], A.indptr[i+1]-1
Out[23]: (4, 5)
In [24]: A.indices[A.indptr[i]:A.indptr[i+1]]
Out[24]: array([2, 3], dtype=int32)
In [25]: A.data[A.indptr[i]:A.indptr[i+1]]
Out[25]: array([5, 6], dtype=int64)
In [26]: A[2, 2], A[2,3]  # check
Out[26]: (5, 6)
```

While the CSR storage method is less intuitive than COO, LIL, or DOK, it is well suited for implementing matrix arithmetic and linear algebra operations. Together with the CSC format, it is, therefore, the main format for use in sparse matrix computations. The CSC format is almost identical to CSR, except that instead of column indices and row pointers, row indices and column pointers are used (i.e., the role of columns and rows is reversed).

## Functions for Creating Sparse Matrices

As shown in the examples earlier in this chapter, one way of constructing sparse matrices is to prepare the data structures for a specific sparse matrix format and pass these to the constructor of the corresponding sparse matrix class. While this method is sometimes suitable, composing sparse matrices from predefined template matrices is often more convenient. The `sp.sparse` module provides a variety of functions for

generating such matrices, for example, `sp.eye` for creating diagonal sparse matrices with ones on the diagonal (optionally offset from the main diagonal), `sp.diags` for creating diagonal matrices with a specified pattern along the diagonal, `sp.kron` for calculating the Kronecker (tensor) product of two sparse matrices, and `bmat`, `vstack`, and `hstack`, for building sparse matrices from sparse block matrices, and by stacking sparse matrices vertically and horizontally, respectively.

For example, in many applications, sparse matrices have a diagonal form. To create a sparse matrix of size 10 × 10 with a main diagonal and an upper and lower diagonal, we can use three calls to `sp.eye`, using the `k` argument to specify the offset from the main diagonal.

```
In [27]: N = 10
In [28]: A = sp.eye(N, k=1) - 2 * sp.eye(N) + sp.eye(N, k=-1)
In [29]: A
Out[29]: <10x10 sparse matrix of type '<class 'numpy.float64'>'
                    with 28 stored elements in Compressed Sparse Row format>
```

By default, the resulting object is a sparse matrix in the CSR format, but using the `format` argument, we can specify any other sparse matrix format. The value of the format argument should be a string such as `'csr'`, `'csc'`, `'lil'`, and so on. All functions for creating sparse matrices in `sp.sparse` accept this argument. For example, in the previous example, we could have produced the same matrix using `sp.diags`, by specifying the pattern `[1, -2, 1]` (the coefficients to the `sp.eye` functions in the previous expression) and the corresponding offsets from the main diagonal `[1, 0, -1]`. If we want the resulting sparse matrix in CSC format, we can set `format='csc'`.

```
In [30]: A = sp.diags([1, -2, 1], [1, 0, -1], shape=[N, N], format='csc')
In [31]: A
Out[31]: <10x10 sparse matrix of type '<class 'numpy.float64'>'
                    with 28 stored elements in Compressed Sparse Column format>
```

The advantages of using sparse matrix formats rather than dense matrices manifest when working with large matrices. Sparse matrices are, by their nature, therefore, large, and hence, it can be challenging to visualize a matrix by, for example, printing its elements in the terminal. Matplotlib provides the `spy` function, a valuable tool for visualizing the structure of a sparse matrix. It is available as a function in the `pyplot` module or as a method for `Axes` instances. Using it on the previously defined `A` matrix, obtains the results shown in Figure 10-1.

```
In [32]: fig, ax = plt.subplots()
    ...: ax.spy(A)
```

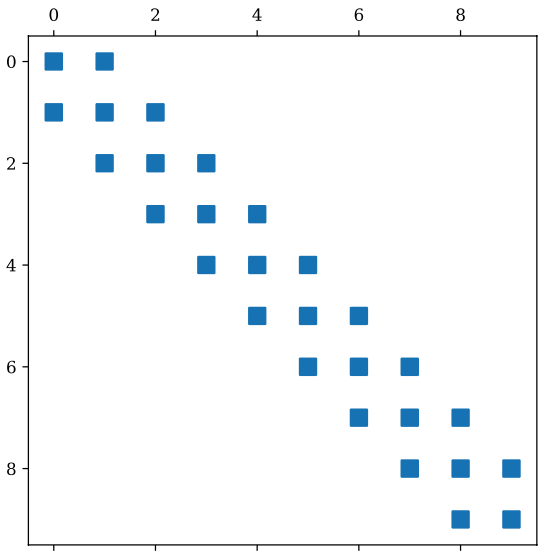


***Figure 10-1.*** *Structure of the sparse matrix with nonzero elements only on the two diagonals closest to the main diagonal and the main diagonal itself*

Sparse matrices are also often associated with tensor product spaces. We can use the `sp.kron` function to compose a sparse matrix from its smaller components for such cases. For example, to create a sparse matrix for the tensor product between A and the matrix

$$B = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

we can use `sp.kron(A, B)`:

```
In [33]: B = sp.diags([1, 1], [-1, 1], shape=[3,3])
In [34]: C = sp.kron(A, B)
In [35]: fig, (ax_A, ax_B, ax_C) = plt.subplots(1, 3, figsize=(12, 4))
    ...: ax_A.spy(A)
    ...: ax_B.spy(B)
    ...: ax_C.spy(C)
```

For comparison, let's plot the sparse matrix structure of *A*, *B*, and *C*; the result is shown in Figure 10-2. For more detailed information on ways to build sparse matrices with the `sp.sparse` module, see its docstring and the "Sparse Matrices" section in the SciPy reference manual.

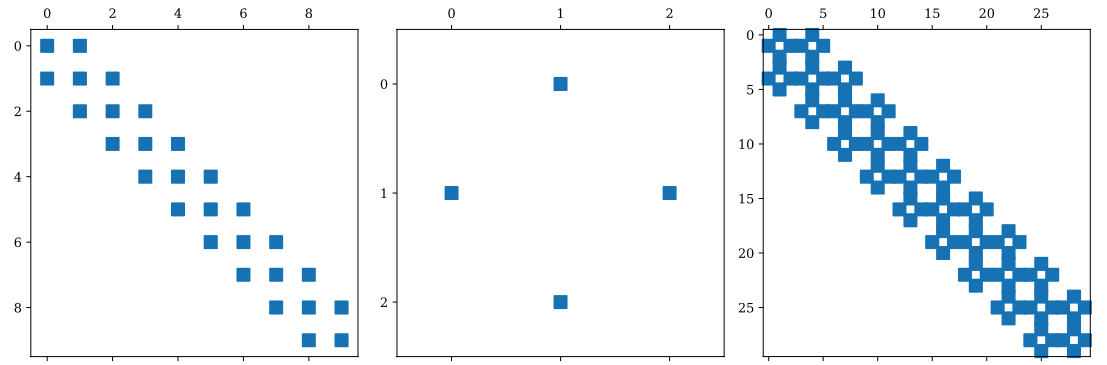

***Figure 10-2.*** *The sparse matrix structures of two matrices, A (left) and B (middle), and their tensor product (right)*

## Sparse Linear Algebra Functions

The main application of sparse matrices is to perform linear algebra operations on large matrices that are intractable or inefficient to treat using dense matrix representations. The SciPy `sparse` module contains a `linalg` submodule that implements many linear algebra routines. Not all linear algebra operations are suitable for sparse matrices. In some cases, the behavior of the sparse matrix version of operations needs to be modified compared to the dense counterparts. Consequently, there are several differences between the sparse linear algebra module `scipy.sparse.linalg` and the dense linear algebra module `scipy.linalg`. For example, the eigenvalue solvers for dense problems typically compute and return all eigenvalues and eigenvectors. This is not manageable for sparse matrices because storing all eigenvectors of a sparse matrix $A$ of size $N \times N$ usually amounts to storing a dense matrix of size $N \times N$. Instead, sparse eigenvalue solvers typically give *a few* eigenvalues and eigenvectors, for example, those with the smallest or largest eigenvalues. In general, sparse matrix methods must retain the sparsity of matrices involved in the computation to be efficient. An example of an operation where the sparsity is usually not retained is the matrix inverse, which should, therefore, be avoided when possible.

## Linear Equation Systems

The most important application of sparse matrices is arguably to solve linear equation systems on the form $Ax = b$, where $A$ is a sparse matrix and $x$ and $b$ are dense vectors. The SciPy `sparse.linalg` module has both direct and iterative solver for this type of problem (`sp.linalg.spsolve`) and methods to factor a matrix $A$, using, for example, $LU$ factorization (`sp.linalg.splu`) and incomplete $LU$ factorization (`sp.linalg.spilu`). For example, consider the problem $Ax = b$ where $A$ is the tridiagonal matrix considered in the preceding text and $b$ is a dense vector filled with negative ones (see Chapter 11 for a physical interpretation of this equation). To solve this problem for the system size $10 \times 10$, first create the sparse matrix `A` and the dense vector `b`.

```
In [36]: N = 10
In [37]: A = sp.diags([1, -2, 1], [1, 0, -1], shape=[N, N], format='csc')
In [38]: b = -np.ones(N)
```

Now, to solve the equation system using the direct solver provided by SciPy, we can use the following.

```
In [39]: x = sp.linalg.spsolve(A, b)
In [40]: x
Out[40]: array([  5.,   9.,  12.,  14.,  15.,  15.,  14.,  12.,   9.,   5.])
```

The solution vector is a dense NumPy array. For comparison, we can also solve this problem using the dense direct solver in NumPy `np.linalg.solve` (or, similarly, using `scipy.linalg.solve`). To use the dense solver, we need to convert the sparse matrix `A` to a dense array using `A.todense()`.

```
In [41]: np.linalg.solve(A.todense(), b)
Out[41]: array([  5.,   9.,  12.,  14.,  15.,  15.,  14.,  12.,   9.,   5.])
```

As expected, the result agrees with what was obtained from the sparse solver. For small problems like this one, there is little to gain from using sparse matrices, but for increasing system size, the merits of using sparse matrices and sparse solvers become apparent. For this particular problem, the threshold system size beyond which sparse methods outperform dense methods is approximately $N = 100$, as shown in Figure 10-3. While the exact threshold varies from problem to problem and hardware and software versions, this behavior is typical for problems where the matrix $A$ is sufficiently sparse.[5]

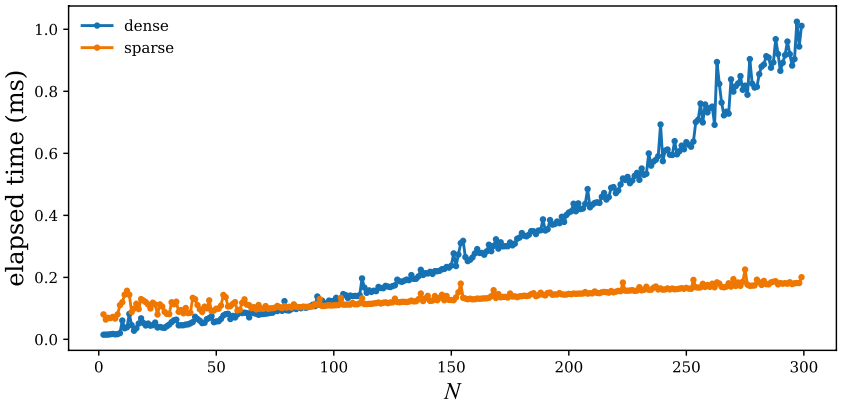


***Figure 10-3.*** *Performance comparison between sparse and dense methods to solve the one-dimensional Poisson problem as a function of problem size*

An alternative to the `spsolve` interface is explicitly computing the LU factorization using `sp.sparse.splu` or `sp.sparse.spilu` (incomplete LU factorization). These functions return an object that contains the $L$ and $U$ factors, and that has a method that solves $LUx = b$ for a given vector $b$. This is particularly useful when the $Ax = b$ must be solved for multiple vectors $b$. For example, the LU factorization of the matrix $A$ used previously is computed using the following.

```
In [42]: lu = sp.linalg.splu(A)
In [43]: lu.L
Out[43]: <10x10 sparse matrix of type '<class 'numpy.float64'>'
                    with 20 stored elements in Compressed Sparse Column format>
```

[5] For a discussion of techniques and methods to optimize Python code, see Chapter 21.

```
In [44]: lu.U
Out[44]: <10x10 sparse matrix of type '<class 'numpy.float64'>'
                    with 20 stored elements in Compressed Sparse Column format>
```

Once the LU factorization is available, we can efficiently solve the equation $LUx = b$ using the `solve` method for the `lu` object.

```
In [45]: x = lu.solve(b)
In [46]: x
Out[46]: array([  5.,   9.,  12.,  14.,  15.,  15.,  14.,  12.,   9.,   5.])
```

An important consideration that arises with sparse matrices is that the LU factorization of $A$ may introduce new nonzero elements in $L$ and $U$ compared to the matrix $A$ and, therefore, make $L$ and $U$ less sparse. Elements that exist in $L$ or $U$ *but not in* $A$ are called fill-ins. The advantage of using sparse matrices may be lost if the number of fill-ins is large. While there is no complete solution to eliminate fill-ins, it is often possible to reduce fill-ins by permuting the rows and columns in A so that the LU factorization takes the form $P_r A P_c = LU$, where $P_r$ and $P_c$ are row and column permutation matrices, respectively. Several such methods for permutations methods are available. The `spsolve`, `splu`, and `spilu` functions all take the `permc_spec` argument, which can take the values `NATURAL`, `MMD_ATA`, `MMD_AT_PLUS_A`, or `COLAMD`, which indicates different permutation methods that are built into these methods. The object returned by `splu` and `spilu` accounts for such permutations, and the permutation vectors are available via the `perm_c` and `perm_r` attributes. Because of these permutations, the product of `lu.L` and `lu.U` is not directly equal to `A`, and to reconstruct `A` from `lu.L` and `lu.U`, we also need to undo the row and column permutations.

```
In [47]: def sp_permute(A, perm_r, perm_c):
    ...:     """ permute rows and columns of A """
    ...:     M, N = A.shape
    ...:     # row permutation matrix
    ...:     Pr = sp.coo_matrix((np.ones(M), (perm_r, np.arange(N)))).tocsr()
    ...:     # column permutation matrix
    ...:     Pc = sp.coo_matrix((np.ones(M), (np.arange(M), perm_c))).tocsr()
    ...:     return Pr.T * A * Pc.T
In [48]: lu.L * lu.U - A  # != 0
Out[48]: <10x10 sparse matrix of type '<class 'numpy.float64'>'
                    with 8 stored elements in Compressed Sparse Column format>
In [49]: sp_permute(lu.L * lu.U, lu.perm_r, lu.perm_c) - A  # == 0
Out[49]: <10x10 sparse matrix of type '<class 'numpy.float64'>'
                    with 0 stored elements in Compressed Sparse Column format>
```

By default, the direct sparse linear solver in SciPy uses the SuperLU[6] package. An alternative sparse matrix solver that can also be used in SciPy is the UMFPACK 7 package. However, this package is not bundled with SciPy and requires the installation of the `scikit-umfpack` Python library. If `scikit-umfpack` is available, and if the `use_umfpack` argument to the `sp.linalg.spsolve` function is `True`, the UMFPACK is used instead of SuperLU. Whether SuperLU or UMFPACK gives better performance varies from problem to problem, so it is worth having both installed and tested for any given problem.

The `sp.spsolve` function is an interface to direct solvers, which internally performs matrix factorization. An alternative approach is to use iterative methods that originate in optimization. The SciPy `sparse.linalg` module contains several functions for the iterative solution of sparse linear problems: for example, `bicg`

[6] http://crd-legacy.lbl.gov/~xiaoye/SuperLU/

(biconjugate gradient method), `bicgstab` (biconjugate gradient stabilized method), `cg` (conjugate gradient), `gmres` (generalized minimum residual), and `lgmres` (loose generalized minimum residual method). These functions (and a few others) can solve the problem $Ax = b$ by calling the function with `A` and `b` as arguments. They all return a tuple `(x, info)` where `x` is the solution, and `info` contains additional information about the solution process (`info=0` indicates success, and it is positive for convergence error and negative for input error). The following is an example.

```
In [50]: x, info = sp.linalg.bicgstab(A, b)
In [51]: x
Out[51]: array([  5.,   9.,  12.,  14.,  15.,  15.,  14.,  12.,   9.,   5.])
In [52]: x, info = sp.linalg.lgmres(A, b)
In [53]: x
Out[53]: array([  5.,   9.,  12.,  14.,  15.,  15.,  14.,  12.,   9.,   5.])
```

In addition, each iterative solver takes its solver-dependent arguments. See the docstring for each function for details. Iterative solvers may have an advantage over direct solvers for large problems, where direct solvers may require excessive memory usage due to undesirable fill-ins. In contrast, iterative solvers only require the evaluation of sparse matrix-vector multiplications and, therefore, do not suffer from fill-in problems. On the other hand, they might have slow convergence for many problems, especially if not adequately preconditioned.

## Eigenvalue Problems

Sparse eigenvalue and singular-value problems can be solved using the `sp.linalg.eigs` and `sp.linalg.svds` functions, respectively. For real symmetric or complex Hermitian matrices, the eigenvalues (which in this case are real) and eigenvectors can also be computed using `sp.linalg.eigsh`. These functions do not compute all eigenvalues or singular values but rather compute a given number of eigenvalues and vectors (the default is six). Using the keyword argument `k` with these functions, we can define how many eigenvalues and vectors should be computed. We can specify the `k` values to be computed using the `which` keyword argument. The options for `eigs` are the largest magnitude `LM`, smallest magnitude `SM`, largest real part `LR`, smallest real part `SR`, largest imaginary part `LI`, and smallest imaginary part `SI`. For `svds`, only `LM` and `SM` are available.

For example, to compute the lowest four eigenvalues for the sparse matrix of the one-dimensional Poisson problem (of system size 10x10), we can use `sp.linalg.eigs(A, k=4, which='LM')`.

```
In [54]: N = 10
In [55]: A = sp.diags([1, -2, 1], [1, 0, -1], shape=[N, N], format='csc')
In [56]: evals, evecs = sp.linalg.eigs(A, k=4, which='LM')
In [57]: evals
Out[57]: array([-3.91898595+0.j, -3.68250707+0.j, -3.30972147+0.j, -2.83083003+0.j])
```

The return value of `sp.linalg.eigs` (and `sp.linalg.eigsh`) is a tuple `(evals, evecs)` whose first element is an array of eigenvalues (`evals`), and the second element is an array (`evecs`) of shape $N \times k$, whose columns are the eigenvectors corresponding to the $k$ eigenvalues in `evals`. Thus, expect the dot product between `A` and a column in `evecs` to be equal to the same column in `evecs` scaled by the corresponding eigenvalue in `evals`. We can directly confirm that this is indeed the case.

```
In [58]: np.allclose(A.dot(evecs[:,0]), evals[0] * evecs[:,0])
Out[58]: True
```

For this particular example, the sparse matrix A is symmetric, so instead of `sp.linalg.eigs`, we could use `sp.linalg.eigsh`, and in doing so, obtain an eigenvalue array with real-valued elements.

```
In [59]: evals, evecs = sp.linalg.eigsh(A, k=4, which='LM')
In [60]: evals
Out[60]: array([-3.91898595, -3.68250707, -3.30972147, -2.83083003])
```

Changing the `which='LM'` argument (largest magnitude) to `which='SM'` (smallest magnitude), obtains a different set of eigenvalues and vectors (those with the smallest magnitude).

```
In [61]: evals, evecs = sp.linalg.eigs(A, k=4, which='SM')
In [62]: evals
Out[62]: array([-0.08101405+0.j, -0.31749293+0.j, -0.69027853+0.j, 1.1691699+0.j])
In [63]: np.real(evals).argsort()
Out[63]: array([3, 2, 1, 0])
```

Note that although we requested and obtained the four eigenvalues with the smallest magnitude in the previous example, those eigenvalues and vectors are not necessarily sorted within each other (although they are in this case). Obtaining sorted eigenvalues is often desirable, and this is easily achieved with a small but convenient wrapper function that sorts the eigenvalues using NumPy's `argsort` method. Let's give such a function, `sp_eigs_sorted`, which returns the eigenvalues and eigenvectors sorted by the real part of the eigenvalue.

```
In [64]: def sp_eigs_sorted(A, k=6, which='SR'):
    ...:     """ compute and return eigenvalues sorted by the real part """
    ...:     evals, evecs = sp.linalg.eigs(A, k=k, which=which)
    ...:     idx = np.real(evals).argsort()
    ...:     return evals[idx], evecs[idx]
In [65]: evals, evecs = sp_eigs_sorted(A, k=4, which='SM')
In [66]: evals
Out[66]: array([-1.16916997+0.j, -0.69027853+0.j, -0.3174929+0.j, -0.0810140+0.j])
```

As a less trivial example using `sp.linalg.eigs` and the wrapper function `sp_eigs_sorted`, consider the spectrum of lowest eigenvalues of the linear combination $(1-x)M_1 + xM_2$ of random sparse matrices $M_1$ and $M_2$. We can use the `sp.rand` function to generate two random sparse matrices, and by repeatedly using `sp_eigs_sorted` to find the smallest 25 eigenvalues of the $(1-x)M_1 + xM_2$ matrix for different values of $x$, we can build a matrix (`evals_mat`) that contains the eigenvalues as a function of $x$.

The following uses 50 values of $x$ in the interval [0, 1].

```
In [67]: N = 100
In [68]: x_vec = np.linspace(0, 1, 50)
In [69]: M1 = sp.rand(N, N, density=0.2)
In [70]: M2 = sp.rand(N, N, density=0.2)
In [71]: evals_mat = np.array(
    ...:     [sp_eigs_sorted((1-x)*M1 + x*M2, k=25)[0] for x in x_vec])
```

Once the matrix `evals_mat` of eigenvalues as a function of $x$ is computed, we can plot the eigenvalue spectrum. The result is shown in Figure 10-4, which is a complicated eigenvalue spectrum due to the randomness of the matrices $M_1$ and $M_2$.

```
In [72]: fig, ax = plt.subplots(figsize=(8, 4))
    ...: for idx in range(evals_mat.shape[1]):
    ...:     ax.plot(x_vec, np.real(evals_mat[:,idx]), lw=0.5)
    ...: ax.set_xlabel(r"$x$", fontsize=16)
    ...: ax.set_ylabel(r"eig.vals. of $(1-x)M_1+xM_2$", fontsize=16)
```

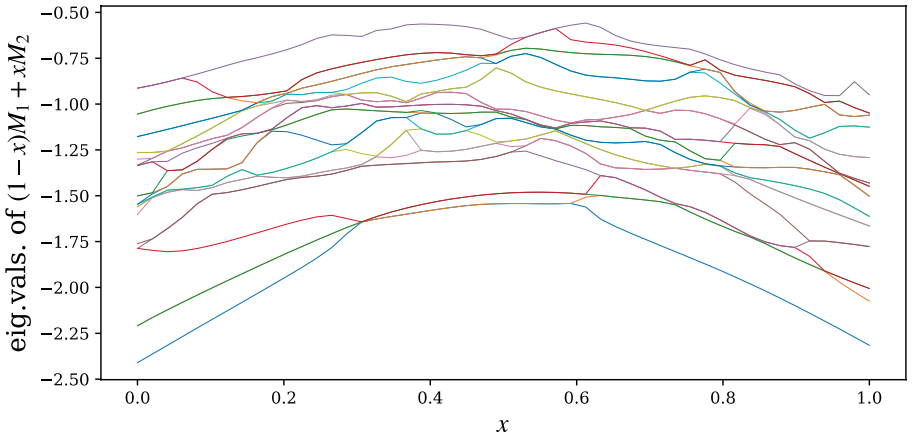


***Figure 10-4.*** *The spectrum of the lowest 25 eigenvalues of the sparse matrix $(1-x)M_1+xM_2$, as a function of x, where $M_1$ and $M_2$ are two random matrices*

## Graphs and Networks

Representing graphs as adjacency matrices is another important application of sparse matrices. In an adjacency matrix, an element describes which nodes in a graph are connected to each other. Consequently, the adjacency matrix is sparse if each node is only connected to a small set of other nodes. The `csgraph` module in the SciPy `sparse` module provides functions for processing such graphs, including methods for traversing a graph using different methods (e.g., breadth-first and depth-first traversals) and for computing shortest paths between nodes in a graph, and so on. For more information about this module, refer to its docstring: `help(sp.csgraph)`.

For a more comprehensive framework for working with graphs, there is the NetworkX Python library. It provides utilities for creating and manipulating undirected and directed graphs and implements many graph algorithms, such as finding minimum paths between nodes in a graph. Here, assume that the NetworkX library is imported under the name `nx`. Using this library, we can, for example, create an undirected graph by initiating an object of the `nx.Graph` class. Any hashable Python object can be stored as nodes in a `graph` object, which makes it a very flexible data structure. However, in the following examples, only use graph objects with integers and strings as node labels. Table 10-2 provides a summary of functions for creating graphs and adding nodes and edges to graph objects.

***Table 10-2.*** *Summary of Objects and Methods for Basic Graph Construction Using NetworkX*

| Object/Method | Description |
|---|---|
| `nx.Graph` | A class for representing undirected graphs. |
| `nx.DiGraph` | A class for representing directed graphs. |
| `nx.MultiGraph` | A class for representing undirected graphs with support for multiple edges. |
| `nx.MultiDiGraph` | A class for representing directed graphs with support for multiple edges. |
| `add_node` | Adds a node to the graph. Expects a node label (e.g., a string or a hashable object) as an argument. |
| `add_nodes_from` | Adds multiple nodes. Expects a list (or iterable) of node labels as arguments. |
| `add_edge` | Adds an edge. Expects two node arguments as arguments and creates an edge between those nodes. |
| `add_edges_from` | Adds multiple edges. Expects a list (or iterable) of tuples of node labels. |
| `add_weighted_edges_from` | Adds multiple edges with weight factors. Expects a list (or iterable) of tuples each containing two node labels and the weight factor. |

For example, we can create a simple graph with node data of integers using `nx.Graph()`, and the `add_node` method, or `add_nodes_from`, to add multiple nodes in one go. The `nodes` method returns an iterator object for the nodes, called a `NodeView`.

```
In [73]: g = nx.Graph()
In [74]: g.add_node(1)
In [75]: g.nodes()
Out[75]: NodeView((1,))
In [76]: g.add_nodes_from([3, 4, 5])
In [77]: g.nodes()
Out[77]: NodeView((1, 3, 4, 5))
```

To connect nodes, we can add edges using `add_edge`. Pass the labels of the two nodes we want to connect as arguments. To add multiple edges, we can use `add_edges_from` and pass to it a list of tuples of nodes to connect. The `edges` method returns an iterator object for the edges called `EdgeView`.

```
In [78]: g.add_edge(1, 2)
In [79]: g.edges()
Out[79]: EdgeView([(1, 2)])
In [80]: g.add_edges_from([(3, 4), (5, 6)])
In [81]: g.edges()
Out[81]: EdgeView([(1, 2), (3, 4), (5, 6)])
```

To represent edges between nodes with weights associated with them (e.g., a distance), we can use `add_weighted_edges_from`, to which we pass a list of tuples containing the weight factor for each edge in addition to the two nodes. When calling the `edges` method, we can additionally give argument `data=True` to indicate that the edge data should also be included in the resulting view.

```
In [82]: g.add_weighted_edges_from([(1, 3, 1.5), (3, 5, 2.5)])
In [83]: g.edges(data=True)
Out[83]: EdgeDataView([(1, 2, {}),
                       (1, 3, {'weight': 1.5}),
                       (3, 4, {}),
                       (3, 5, {'weight': 2.5}),
                       (5, 6, {})])
```

Note that if we add edges between nodes that do not yet exist in the graph, they are seamlessly added. For example, add a weighted edge between nodes 6 and 7 in the following code. Node 7 does not previously exist in the graph, but when adding an edge, it is automatically created and added.

```
In [84]: g.add_weighted_edges_from([(6, 7, 1.5)])
In [85]: g.nodes()
Out[85]: NodeView((1, 3, 4, 5, 2, 6, 7))
In [86]: g.edges()
Out[86]: EdgeView([(1, 2), (1, 3), (3, 4), (3, 5), (5, 6), (6, 7)])
```

With these fundamentals in place, we can look at a more complicated graph example. The following builds a graph from a dataset stored in a JSON file called `tokyo-metro.json` (available with the code listings), which we load using the Python standard library module `json`.[7]

```
In [87]: import json
In [88]: with open("tokyo-metro.json") as f:
    ...:     data = json.load(f)
```

The result of loading the JSON file is a dictionary `data` that contains metro line descriptions. Each line has a list of travel times between stations (`travel_times`), possible transfer points to other lines (transfer), and the line color.

```
In [89]: data.keys()
Out[89]: dict_keys(['C', 'T', 'N', 'F', 'Z', 'M', 'G', 'Y', 'H'])
In [90]: data["C"]
Out[90]: {'color': '#149848',
          'transfers': [['C3', 'F15'], ['C4', 'Z2'], ...],
          'travel_times': [['C1', 'C2', 2], ['C2', 'C3', 2], ...]}
```

Here the format of the `travel_times` list is `[['C1', 'C2', 2], ['C2', 'C3', 2], ...]`, indicating that it takes 2 minutes to travel between the stations C1 and C2, and 2 minutes to travel between C2 and C3, and so on. The format of the `transfers` list is `[['C3', 'F15'], ...]`, indicating that it is possible to transfer from the C line to the F line at station C3 to station F15. The `travel_times` and `transfers` are directly suitable for feeding to `add_weighed_edges_from` and `add_edges_from`, and we can easily create a graph for representing the metro network by iterating over each metro line dictionary and calling these methods.

```
In [91]: g = nx.Graph()
    ...: for line in data.values():
    ...:     g.add_weighted_edges_from(line["travel_times"])
    ...:     g.add_edges_from(line["transfers"])
```

[7] For more information about the JSON format and the `json` module, see Chapter 18.

The line transfer edges do not have edge weights, so let's first mark all transfer edges by adding a new Boolean attribute `transfer` to each edge.

```
In [92]: for n1, n2 in g.edges():
    ...:     g[n1][n2]["transfer"] = "weight" not in g[n1][n2]
```

Next, for plotting purposes, create two lists of edges containing transfer edges and on-train edges, and also create a list with colors corresponding to each node in the network.

```
In [93]: on_foot = [e for e in g.edges() if g.get_edge_data(*e)["transfer"]]
In [94]: on_train = [e for e in g.edges () if not g.get_edge_data(*e)["transfer"]]
In [95]: colors = [data[n[0].upper()]["color"] for n in g.nodes()]
```

To visualize the graph, we can use the Matplotlib-based drawing routines in the Networkx library: use `nx.draw` to draw each node, `nx.draw_networkx_labels` to draw the labels to the nodes, and `nx.draw_network_edges` to draw the edges. Call `nx.draw_network_edges` twice, with the edge lists for transfers (`on_foot`) and on-train (`on_train`) connections, and color the links blue and black, respectively, using the `edge_color` argument. The `pos` argument to the drawing functions determines the layout of the graph. Use the `graphviz_layout` function from `networkx.drawing.nx_agraph` to lay out the nodes. All drawing functions also accept a Matplotlib axes instance via the `ax` argument. The resulting graph is shown in Figure 10-5.

```
In [96]: fig, ax = plt.subplots(1, 1, figsize=(14, 10))
    ...: pos = nx.drawing.nx_agraph.graphviz_layout(g, prog="neato")
    ...: nx.draw(g, pos, ax=ax, node_size=200, node_color=colors)
    ...: nx.draw_networkx_labels(g, pos=pos, ax=ax, font_size=6)
    ...: nx.draw_networkx_edges(g, pos=pos, ax=ax, edgelist=on_train, width=2)
    ...: nx.draw_networkx_edges(
    ...:     g, pos=pos, ax=ax, edgelist=on_foot, edge_color="blue")
```

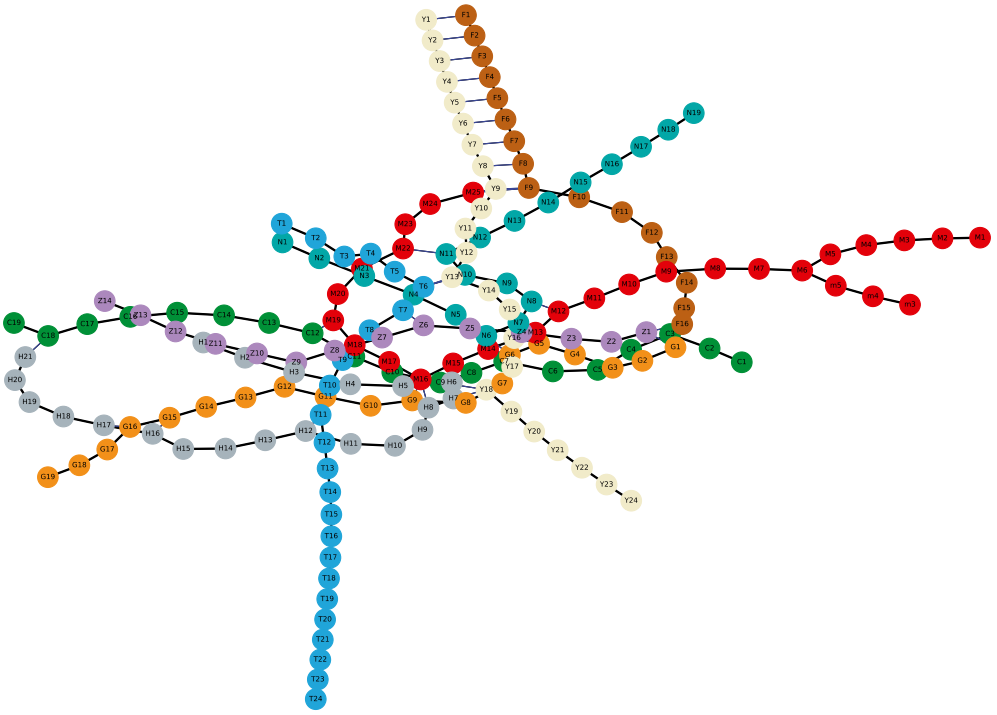

***Figure 10-5.*** *Network graph for the Tokyo Metro stations*

Once the network has been constructed, we can use the many graph algorithms provided by the NetworkX library to analyze the network. For example, to compute the degree (i.e., the number of connections to a node) of each node, we can use the `degree` method (here, the output is truncated at `...` to save space).

```
In [97]: g.degree()
Out[97]: DegreeView({'Y8': 3,  'N18': 2,  'M24': 2,  'G15': 3,  'C18': 3, ... })
```

For this graph, the degree of a node can be interpreted as the number of connections to a station: the more metro lines that connect at a station, the higher the degree of the corresponding node. We can easily search for the most highly connected station in the network using the degree method and the max function to find the highest degree. Next, iterate over the result of the `degree` method and select the nodes with the maximal degree (which is 6 in this network).

```
In [98]: d_max = max(d for (n, d) in g.degree())
In [99]: [(n, d) for (n, d) in g.degree() if d == d_max]
Out[99]: [('N7', 6), ('G5', 6), ('Y16', 6), ('M13', 6), ('Z4', 6)]
```

The result shows that the most highly connected stations are station numbers 7 on the N line, 5 on the G line, and so on. All these lines intercept at the same station (the Nagatachou station). We can also compute the closest path between two points in the network using `nx.shortest_path`. For example, the following finds the optimal traveling route (assuming no waiting time and instantaneous transfer) for traveling between Y24 and C19.

```
In [100]: p = nx.shortest_path(g, "Y24", "C19")
In [101]: p
Out[101]: ['Y24', 'Y23', 'Y22', 'Y21', 'Y20', 'Y19', 'Y18', 'C9', 'C10', 'C11',
           'C12', 'C13', 'C14', 'C15', 'C16', 'C17', 'C18', 'C19']
```

Given a path on this form, we can also directly evaluate the travel time by summing up the weight attributes of neighboring nodes in the path.

```
In [102]: np.sum([g[p[n]][p[n+1]]["weight"]
     ...:         for n in range(len(p)-1) if "weight" in g[p[n]][p[n+1]]])
Out[102]: 35
```

The result suggests it takes 35 minutes to travel from Y24 to C19. Since the transfer nodes do not have a weight associated with them, the train transfers are effectively assumed to be instantaneous. It may be reasonable to assume that a train transfer takes about 5 minutes. To consider this in the shortest path and travel time computation, we can update the transfer nodes and add a weight of 5 to each. To do this, create a copy of the graph using the `copy` method, iterate through the edges, and update those with the `transfer` attribute set to `True`.

```
In [103]: h = g.copy()
In [104]: for n1, n2 in h.edges():
     ...:     if h[n1][n2]["transfer"]:
     ...:         h[n1][n2]["weight"] = 5
```

Recomputing the path and the traveling time with the new graph gives a more realistic estimate of the traveling time.

```
In [105]: p = nx.shortest_path(h, "Y24", "C19")
In [106]: p
Out[106]: ['Y24', 'Y23', 'Y22', 'Y21', 'Y20', 'Y19', 'Y18', 'C9', 'C10', 'C11', 'C12',
'C13', 'C14', 'C15', 'C16', 'C17', 'C18', 'C19']
In [107]: np.sum([h[p[n]][p[n+1]]["weight"] for n in range(len(p)-1)])
Out[107]: 40
```

With this method, we can compute the optimal path and travel time between arbitrary nodes in the network. For example, we can compute the shortest path and traveling time between Z1 and H16 (32 minutes).

```
In [108]: p = nx.shortest_path(h, "Z1", "H16")
In [109]: np.sum([h[p[n]][p[n+1]]["weight"] for n in range(len(p)-1)])
Out[109]: 32
```

The NetworkX representation of a graph can be converted to an adjacency matrix in the form of a SciPy sparse matrix using the `nx.to_scipy_sparse_array`, after which we can also analyze the graph with the routines in the `sp.csgraph` module. As an example of this, let's convert the Tokyo Metro graph to an adjacency matrix and compute its reverse Cuthill-McKee ordering (using `sp.csgraph.reverse_cuthill_mckee`, which is a reordering that reduces the maximum distance of the matrix elements from the diagonal) and permute the matrix with this ordering. Plot the result of both matrices using Matplotlib's `spy` function, as shown in Figure 10-6.

```
In [110]: A = nx.to_scipy_sparse_array(g)
In [111]: A
Out[111]: <184x184 sparse matrix of type '<class 'numpy.int64'>'
                    with 486 stored elements in Compressed Sparse Row format>
In [112]: perm = sp.csgraph.reverse_cuthill_mckee(A)
In [113]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 4))
     ...: ax1.spy(A, markersize=2)
     ...: ax2.spy(sp_permute(A, perm, perm), markersize=2)
```

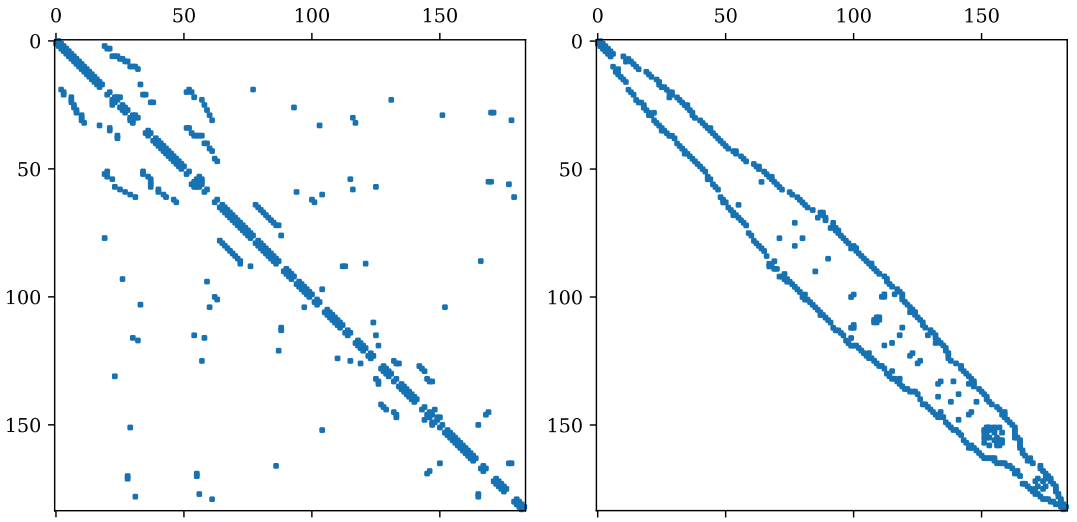


***Figure 10-6.*** *The adjacency matrix of the Tokyo metro graph (left) and the same after RCM ordering (right)*

# Summary

This chapter introduced common methods of storing sparse matrices and reviewed how these can be represented using the sparse matrix classes in the SciPy `sparse` module. It also reviewed the sparse matrix construction functions in the SciPy `sparse` module and the sparse linear algebra routines in `sparse.linalg`. To complement the linear algebra routines built into SciPy, the chapter briefly discussed the `scikit.umfpack` extension package, which makes the UMFPACK solver available to SciPy. The sparse matrix library in SciPy is versatile and very convenient to work with. It also offers good performance because it uses efficient low-level libraries for linear algebra routines (SuperLU or UMFPACK). For large-scale problems that require parallelization to distribute the workload to multiple cores or even multiple computers, the PETSc and Trilinos frameworks, which both have Python interfaces, provide routes for using sparse matrices and sparse linear algebra with Python in high-performance applications. Graph representations and processing using the SciPy `sparse.csgraph` and NetworkX libraries were also introduced.

# Further Reading

A good and accessible introduction to sparse matrices and direct solvers for sparse linear equation systems is given in *Direct Methods for Sparse Linear Systems* by T. Davis (SIAM, 2006). A detailed discussion of sparse matrices and methods is provided in *Numerical Recipes in C: The Art of Scientific Computing* by W. H. Press (Cambridge University Press, 2007). For a thorough introduction to network and graph theory, see *Networks: An Introduction* by M. Newman (Oxford, 2010).